1.) Consider the following series of address references, given as byte addresses:

4, 16, 32, 20, 80, 68, 76, 224, 36, 44, 16, 172, 20, 24, 36, 68

Label each reference as a hit or miss, and show the final cache contents, for each of the following caches. Assume LRU replacement (where appropriate).

a.) Direct-mapped, 16x 4-byte blocks.

b.) Direct-mapped, 16-byte blocks, total size of 64 bytes.

c.) Two-way set associative, 4-byte blocks, total size of 64 bytes.

d.) Fully associative, 4-byte blocks, total size of 64 bytes.

2.) Compute the total number of bits needed to implement each of the caches from problem 1. This number is different from the size of the cache, which usually refers to the number of bytes of data stored in the cache. The total number of bits needed to implement the cache represents the total amount of memory needed for storing all the data, tags, and valid bits. Note that you can assume 32 bit addresses.

3.) The code below is the toUpper example. Figure out the prediction accuracy for each of the three branch instructions below (individually) with a 1-bit predictor, as well as for the first branch (CBZ) with a 2-bit predictor. Thus, your answer will be 4 percentages. You can assume:

- The code has been executing for a long time, so the predictors are “warm”.
- The predictors do not conflict in the branch history table.
- The string is 20 characters long (including the final null).
- Each of the 19 non-null characters in the string are randomly distributed in the range 1-255.
- Characters are encoded in ASCII code – you can find those values in the book or on-line.

```c
// string is a pointer held at Memory[100].
// X0=index, 'A' = 65, 'a' = 97, 'z' = 122
LDUR X0, [X31, #100]  // index = string

LOOP:
  LDURB X1, [X0, #0]     // load byte *index
  CBZ X1, END            // exit if *index == 0
  CMPI X1, #97           // is *index < 'a'?
    B.LT NEXT            // don't change if < 'a'
  CMPI X1, #122          // is *index > 'z'? '
    B.GT NEXT           // don't change if > 'z'
  SUBI X1, X1, #32       // X1 = *index + ('A' - 'a')
  STURB X1, [X0, #0]     // *index = new value;

NEXT:
  ADDI X0, X0, 1         // index++;        
  B LOOP                 // continue the loop

END:
```